REACT: The Riskmap Evaluation and Coordination Terminal

by

Abraham Quintero

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2019

Author .....
Department of Electrical Engineering and Computer Science
August 20, 2019

Certified by .....
Miho Mazereeuw
Associate Professor
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Chairman, Department Committee on Graduate Theses

REACT: The Riskmap Evaluation and Coordination Terminal

by

Abraham Quintero

Submitted to the Department of Electrical Engineering and Computer Science
on August 20, 2019, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

Disaster information systems use state of the art techniques in order to mitigate damages from natural and man made hazards. Developed countries utilize networks of advanced sensors and ahead of time mapping in order to facilitate emergency responses; however, such systems are not available in developing countries, which face the highest risk from disasters. This work seeks to use novel machine learning techniques to fully utilize crowdsourced social media reports gathered using the Riskmap system. This work establishes the motivation for using citizens as sensors and analyzing this noisy data using machine learning. It also reviews different machine learning techniques for disaster mitigation. Finally a novel ensemble learning model is presented that can accurately predict large flood events from crowdsourced data.

Thesis Supervisor: Miho Mazereeuw
Title: Associate Professor

Acknowledgments

To Aditya and Miho- not one page of this thesis could have been written without your help and your guidance. Thank you so much for your patience and your expertise.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Flooding is the most common natural disaster in the world [2]. Flood related deaths account for half of all deaths from natural disasters. Although flooding impacts both developed and developing countries, developing nations face much worse consequences as a result of flooding since they lack resources to adequately mitigate disasters. Unregulated urbanization, rising population and climate change all contrive to increase the rate at which floods occur in developing megacities; furthermore, there is little data about these disasters [1]. Data scarcity makes it hard to pinpoint where to direct aid during disasters and where to make infrastructure improvements after disasters [4].

Government and NGOs work together to mitigate damages from flooding. Citizens look for relevant flood information and try to reduce their risk. Information is at the core of this interaction; however, data scarcity makes it hard for emergency personnel to optimize their use of resources, while citizens have an abundance of information about their surroundings but must be careful not to trust incorrect or outdated information about broader areas [3]. The natural solution is for citizens on social media to submit real time reports to the Emergency Operations Center, which is tasked with using those reports to inform citizens. There is one problem with this solution, in times of crisis EOCs can suffer from information overload when they are presented with too much information.

The REACT system uses novel machine learning and human computer interaction

research to reduce information overload in EOCs, thereby decreasing disaster response time. REACT learns how Emergency Operations Centers (EOCs) classify the severity of flood events given citizen submitted reports. REACT trains itself through a gamified simulation of a disaster event. During a real disaster, REACT digests social media reports and estimates how severely an event is impacting different areas of a city and thereby helps EOCs to respond in the best manner possible.

1.1 History of Disaster Informatics

1.2.

1.2 The Riskmap System

In order to perform a sequence of floating point operations, a normal FPU performs many redundant internal shifts and normalizations in the process of performing a sequence of operations. However, if a compiler can decompose the floating point operations it needs down to the lowest level, it then can optimize away many of these redundant operations.

If there is some additional hardware support specifically for micro-optimization, there are additional optimizations that can be performed. This hardware support entails extra "guard bits" on the standard floating point formats, to allow several unnormalized operations to be performed in a row without the loss information[1]. A discussion of the mathematics behind unnormalized arithmetic is in appendix **??**.

The optimizations that the compiler can perform fall into several categories:

1.2.1 Motivation for crowdsourced data

When more than two multiplications are performed in a row, the intermediate normalization of the results between multiplications can be eliminated. This is because with each multiplication, the mantissa can become denormalized by at most one bit.

[1]A description of the floating point format used is shown in figures **??** and **??**.

If there are guard bits on the mantissas to prevent bits from "falling off" the end during multiplications, the normalization can be postponed until after a sequence of several multiplies[2].

As you can see, the intermediate results can be multiplied together, with no need for intermediate normalizations due to the guard bit. It is only at the end of the operation that the normalization must be performed, in order to get it into a format suitable for storing in memory[3].

1.2.2 Block Exponent

In a unoptimized sequence of additions, the sequence of operations is as follows for each pair of numbers (m_1,e_1) and (m_2,e_2).

1. Compare e_1 and e_2.
2. Shift the mantissa associated with the smaller exponent $|e_1 - e_2|$ places to the right.
3. Add m_1 and m_2.
4. Find the first one in the resulting mantissa.
5. Shift the resulting mantissa so that normalized
6. Adjust the exponent accordingly.

Out of 6 steps, only one is the actual addition, and the rest are involved in aligning the mantissas prior to the add, and then normalizing the result afterward. In the block exponent optimization, the largest mantissa is found to start with, and all the mantissa's shifted before any additions take place. Once the mantissas have been

[2]Using unnormalized numbers for math is not a new idea; a good example of it is the Control Data CDC 6600, designed by Seymour Cray. [**?**] The CDC 6600 had all of its instructions performing unnormalized arithmetic, with a separate `NORMALIZE` instruction.

[3]Note that for purposed of clarity, the pipeline delays were considered to be 0, and the branches were not delayed.

shifted, the additions can take place one after another[4]. An example of the Block Exponent optimization on the expression X = A + B + C is given in figure **??**.

1.3 Integer optimizations

As well as the floating point optimizations described above, there are also integer optimizations that can be used in the μFPU. In concert with the floating point optimizations, these can provide a significant speedup.

1.3.1 Conversion to fixed point

Integer operations are much faster than floating point operations; if it is possible to replace floating point operations with fixed point operations, this would provide a significant increase in speed.

This conversion can either take place automatically or or based on a specific request from the programmer. To do this automatically, the compiler must either be very smart, or play fast and loose with the accuracy and precision of the programmer's variables. To be "smart", the computer must track the ranges of all the floating point variables through the program, and then see if there are any potential candidates for conversion to floating point. This technique is discussed further in section **??**, where it was implemented.

The other way to do this is to rely on specific hints from the programmer that a certain value will only assume a specific range, and that only a specific precision is desired. This is somewhat more taxing on the programmer, in that he has to know the ranges that his values will take at declaration time (something normally abstracted away), but it does provide the opportunity for fine-tuning already working code.

Potential applications of this would be simulation programs, where the variable represents some physical quantity; the constraints of the physical system may provide bounds on the range the variable can take.

[4]This requires that for n consecutive additions, there are $\log_2 n$ high guard bits to prevent overflow. In the μFPU, there are 3 guard bits, making up to 8 consecutive additions possible.

1.3.2 Small Constant Multiplications

One other class of optimizations that can be done is to replace multiplications by small integer constants into some combination of additions and shifts. Addition and shifting can be significantly faster than multiplication. This is done by using some combination of

$$\begin{aligned} a_i &= a_j + a_k \\ a_i &= 2a_j + a_k \\ a_i &= 4a_j + a_k \\ a_i &= 8a_j + a_k \\ a_i &= a_j - a_k \\ a_i &= a_j \ll m\text{shift} \end{aligned}$$

instead of the multiplication. For example, to multiply s by 10 and store the result in r, you could use:

$$\begin{aligned} r &= 4s + s \\ r &= r + r \end{aligned}$$

Or by 59:

$$\begin{aligned} t &= 2s + s \\ r &= 2t + s \\ r &= 8r + t \end{aligned}$$

Similar combinations can be found for almost all of the smaller integers[5]. [**?**]

[5]This optimization is only an "optimization", of course, when the amount of time spent on the shifts and adds is less than the time that would be spent doing the multiplication. Since the time costs of these operations are known to the compiler in order for it to do scheduling, it is easy for the compiler to determine when this optimization is worth using.

1.4 Other optimizations

1.4.1 Low-level parallelism

The current trend is towards duplicating hardware at the lowest level to provide parallelism[6]

Conceptually, it is easy to take advantage to low-level parallelism in the instruction stream by simply adding more functional units to the μFPU, widening the instruction word to control them, and then scheduling as many operations to take place at one time as possible.

However, simply adding more functional units can only be done so many times; there is only a limited amount of parallelism directly available in the instruction stream, and without it, much of the extra resources will go to waste. One process used to make more instructions potentially schedulable at any given time is "trace scheduling". This technique originated in the Bulldog compiler for the original VLIW machine, the ELI-512. [**?**, **?**] In trace scheduling, code can be scheduled through many basic blocks at one time, following a single potential "trace" of program execution. In this way, instructions that *might* be executed depending on a conditional branch further down in the instruction stream are scheduled, allowing an increase in the potential parallelism. To account for the cases where the expected branch wasn't taken, correction code is inserted after the branches to undo the effects of any prematurely executed instructions.

1.4.2 Pipeline optimizations

In addition to having operations going on in parallel across functional units, it is also typical to have several operations in various stages of completion in each unit. This pipelining allows the throughput of the functional units to be increased, with no increase in latency.

[6]This can been seen in the i860; floating point additions and multiplications can proceed at the same time, and the RISC core be moving data in and out of the floating point registers and providing flow control at the same time the floating point units are active. [**?**]

There are several ways pipelined operations can be optimized. On the hardware side, support can be added to allow data to be recirculated back into the beginning of the pipeline from the end, saving a trip through the registers. On the software side, the compiler can utilize several tricks to try to fill up as many of the pipeline delay slots as possible, as seendescribed by Gibbons. [**?**]

Appendix A

Tables

Table A.1: Armadillos

Armadillos	are
our	friends

Appendix B

Figures

Figure B-1: Armadillo slaying lawyer.

Figure B-2: Armadillo eradicating national debt.

Bibliography

[1] F. K. S. Chan, C. Joon Chuah, A. D. Ziegler, M. Dąbrowski, and O. Varis. Towards resilient flood risk management for Asian coastal cities: Lessons learned from Hong Kong and Singapore. *Journal of Cleaner Production*, 187:576–589, June 2018.

[2] Faith Ka Shun Chan, Gordon Mitchell, Olalekan Adekola, and Adrian McDonald. Flood Risk in Asia's Urban Mega-deltas: Drivers, Impacts and Response. *Environment and Urbanization ASIA*, 3(1):41–61, March 2012.

[3] E. L. Quarantelli. Problematical aspects of the information/ communication revolution for disaster planning and research: Ten non-technical issues and questions. *Disaster Prevention and Management; Bradford*, 6(2):94–106, 1997.

[4] Irfan Ahmad Rana and Jayant K. Routray. Multidimensional Model for Vulnerability Assessment of Urban Flooding: An Empirical Study in Pakistan. *International Journal of Disaster Risk Science; Heidelberg*, 9(3):359–375, September 2018.